

DVLoc For Code Generators

Programmer's Guide

January 2001

Document Number 748283.003

World Wide Web: http://developer.intel.com

Version	Version History	Date
-003	Updated to use the Itanium™ trademark	January 2001

THIS DOCUMENT IS PROVIDED “AS IS” WITH NO WARRANTIES WHATSOEVER, INCLUDING ANY WARRANTY OF MERCHANTABILITY, NONINFRINGEMENT, FITNESS FOR ANY PARTICULAR PURPOSE, OR ANY WARRANTY OTHERWISE ARISING OUT OF ANY PROPOSAL, SPECIFICATION OR SAMPLE.

Information in this document is provided in connection with Intel products. No license, express or implied, by estoppel or otherwise, to any intellectual property rights is granted by this document. Except as provided in Intel's Terms and Conditions of Sale for such products, Intel assumes no liability whatsoever, and Intel disclaims any express or implied warranty, relating to sale and/or use of Intel products including liability or warranties relating to fitness for a particular purpose, merchantability, or infringement of any patent, copyright or other intellectual property right. Intel products are not intended for use in medical, life saving, or life sustaining applications.

Intel may make changes to specifications and product descriptions at any time, without notice.

Designers must not rely on the absence or characteristics of any features or instructions marked "reserved" or "undefined." Intel reserves these for future definition and shall have no responsibility whatsoever for conflicts or incompatibilities arising from future changes to them.

DVLoc and the Scheduling Library may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Current characterized errata are available on request.

Copies of documents which have an order number and are referenced in this document, or other Intel literature, may be obtained by calling 1-800-548-4725, or by visiting Intel's website at http://developer.intel.com/design/litcentr.

Copyright © Intel Corporation, 2000, 2001

*Third-party brands and names are the property of their respective owners.

Contents

Chapter 1 Introduction

About This Manual 1-1
Related Publications 1-2
Notational Conventions 1-2

Chapter 2 Overview

Capabilities 2-1
Environment 2-1
Modes and Usage Flows 2-2
- Abstract Mode 2-2
 - Open DVLoc 2-2
 - Send Clues 2-3
 - Send Block 2-3
 - Close DVLoc 2-4
- Static Mode 2-5
 - Open DVLoc 2-5
 - Send Clues 2-6
 - Send Block 2-6
 - Close DVLoc 2-7
- Dynamic Mode 2-7
 - Open DVLoc 2-7
 - Send Clues 2-7
 - Send Block 2-8
 - Close DVLoc 2-8

Chapter 3 Enumerations and Constants

DVLoc_inspection_mode_t 3-1
DVLoc_inspection_block_t 3-2
DVLoc_env_type_t 3-2
Instr_id_t 3-3
IA64_Resource_t 3-3
EM_Decoder_Reg_Name 3-4
DVLoc_dep_viol_type_t 3-4
DVLoc_dep_viol_id_t 3-4
DVLoc_err_t 3-5
DVLoc_certainty_t 3-7
Dependency_semantics_t 3-8
DVLoc_clue_type_t 3-9
DVLoc_clue_timing_t 3-11
DVLoc_pred_pair_rel_t 3-12
DVLoc_pred_group_rel_t 3-13

Chapter 4 Data Structures

Uint64 4-1
DVLoc_res_info_t 4-1
DVLoc_dep_viol_t 4-2
AbstrInstrInfo - Abstract Instruction 4-3

Chapter 5 Clues

Clue Data Structure 5-1
DVLoc_clue_t 5-1
Predicate Relations Clues 5-3
Single Relation Approach 5-4
DVLoc_pred_pair_t 5-4
DVLoc_pred_group_t 5-5
Set of Relations Approach 5-8
PredInfoArray 5-8
PredInfoArrayEntry 5-9

DVLoc_pdcat_relation_vector_t 5-10
Safe Across Calls Clues 5-12
Rotating Region Size Clue 5-13
st8.spill and ld8.fill Clue 5-14
Register Values Clue 5-15
Skip Dependency Violations Detection Clue 5-17

Chapter 6 Functions

DVLoc_open() 6-1
DVLoc_setenv() 6-2
DVLoc_execution_clue() 6-3
DVLoc_validateAbstrInstr() 6-3
DVLoc_search_dep_viols_abstr_code() 6-5
DVLoc_search_dep_viols() 6-6
DVLoc_search_dep_viols_absolute_loc() 6-7
DVLoc_get_dep_viol() 6-8
DVLoc_get_dep_viol_text() 6-9
DVLoc_close() 6-10

Glossary

Index

Introduction 1

A data dependency violation may cause undefined or unpredictable processor behavior. The objective of DVLoc (Dependency Violation Locator) with the Scheduling Library is to detect dependency violations in Intel® Itanium™ assembly and binary code.

DVLoc can be used in the following stages of the code life cycle:

bundling	When bundling Itanium architecture assembly code, DVLoc may find dependency violations in the code. Stops may be inserted to remove the dependency violations. This is the DVLoc abstract mode.
final	After the Itanium architecture assembly or binary code is created, DVLoc can ensure that the created code does not contain dependency violations. This is the DVLoc static mode.
execution	DVLoc tracks the occurance of dependency violations when executing the code. This is the DVLoc dynamic mode.

About This Manual

The document contains the following chapters:

- Chapter 1, “Introduction”, introduces this document and DVLoc, and lists the manuals referred to in this document.
- Chapter 2, “Overview”, briefly describes the DVLoc modes and usage flows through functions, and explains code partitioning.
- Chapter 3, “Enumerations and Constants”, describes all the enumerations and constants that DVLoc uses.
- Chapter 4, “Data Structures”, describes the DVLoc data structures.
- Chapter 5, “Clues”, describes the DVLoc clue feature.

- Chapter 6, "Functions", describes the various functions in DVLoc's API.

This document also contains a Glossary.

Related Publications

Refer to the following documentation, available at http://developer.intel.com, for more information:

Intel® Itanium™ Architecture Software Developer's Manual

- *Volume 1: Application Architecture*
- *Volume 2: System Architecture*
- *Volume 3: Instruction Set Reference*
- *Volume 4: Itanium Processor Programmer's Guide*

Intel® Itanium™ Architecture Assembly Language Reference Guide

Notational Conventions

`this type style`	Indicates code. Object names may appear in upper and lowercase.
`this type style`	Indicates a parameter.
`[items]`	Indicates that the items enclosed in brackets are optional.
`THIS TYPE STYLE`	Indicates enumeration.
this type style	Indicates a hypertext link.

Overview 2

This chapter explains the DVLoc modes and usage flows, briefly describing DVLoc functions and code partitioning. For a fuller explanation of the functions, see Chapter 6, “Functions”.

Capabilities

DVLoc detects the following types of dependency violations:

- RAW, WAW, and WAR resource dependencies within an instruction group.
- RAW on system resources that exceeds the instruction group boundaries, relevant only in dynamic mode.
- Place in instruction group (Piig) violations, where instructions are incorrectly positioned within an instruction group.

Environment

DVLoc requires the following header files:

`sched_IA64.h`	the central Scheduling Library header file
`sched_enums.h`	containing some enumerations used by the Scheduling Library API

While most of the DVLoc functionality is covered by the `sched_IA64.h` file, for completeness, all the data types, constants, macros, and enumerations relevant for DVLoc usage are explained in this document.

Not all the functions and data structures in `sched_IA64.h` (the Scheduling Library header file) are described in this document.

Include the command-line option –DWINNT when compiling for Windows NT*, to replace all occurrences of Uint64 with the C++ construct __int64.

Modes and Usage Flows

The DVLoc functionality of the Scheduling Library detects dependency violations in Intel® Itanium™ architecture code. DVLoc operates in three modes to gain the functionality described in Chapter 1, "Introduction".

abstract mode — Abstract mode detects dependency violations in code that is not yet encoded as binary. This is useful for code generators, since they can bundle the code while inserting stops, according to reports from DVLoc. See Chapter 4, "Data Structures".

static mode — Static mode detects dependency violations in code that is already encoded as binary. It is difficult to insert stops or instructions to eliminate dependency violations in binary code, so this mode is useful for detection rather than elimination of dependency violations. Potential client applications include linkers, and loaders.

dynamic mode — In dynamic mode, DVLoc assumes that the client application has all the run-time information about the code, including predicate and general register values, occurance of exceptions and interrupts, and whether or not the branch is taken. Potential client applications are Itanium architecture simulators.

The following sections describe the basic usage flow for the three modes.

Abstract Mode

The client application must take the following steps.

Open DVLoc

Call DVLoc_open(DVLOC_INSPECTION_MODE_STATIC_ABSTR) to initialize the DVLoc structures and prepare DVLoc to receive Itanium architecture code.

Send Clues

Pass on all information about the inspection block, prior to sending the block itself, using the function `DVLoc_execution_clue()`. Transfer the information as data structures, called clues, containing the following types of information:

predicate registers relations	A clue for every generation or destruction of some predicate registers relations.
size of rotating registers frame	Allows accurate detection of dependency violations on the Current Frame Marker (CFM) register. A clue is sent for every change in the size of the general registers rotating region size.
accessed place in memory	`ld8.fill` and `st8.spill` instructions allow accurate detection of dependency violations on AR[UNAT]. A clue is sent for every `ld8.fill` and `st8.spill` instruction in the inspection block.

Each clue has a scope, which is the lifetime of the information given by the clue. More information about scopes and usage of clues is in Chapter 5, "Clues".

Send Block

Build the inspection block, which is an array of abstract instructions described in "AbstrInstrInfo - Abstract Instruction" in Chapter 4. Send the inspection block to DVLoc with the function `DVLoc_search_dep_viols_abstr_code()`.

DVLoc searches for dependency violations in the block. The detected dependency violations are stored in a database in DVLoc. Extract the dependency violation information with the function `DVLoc_get_dep_viol()`.

DVLoc searches only the current inspection block. No dependency violation detection is performed between the current inspection block and the previous block. In order to perform accurate and fast dependency violation detection, partition your code into static instruction groups.

The boundaries of the instruction groups must be independent of any run-time conditions. Send each instruction group as a separate inspection block.

Non-conditional instruction group boundaries cause each dependency violation to be fully contained in a single inspection block. Non-conditional instruction group boundaries include one or both of the following:

- stop
- unconditional branch

Boundaries set by conditional branches such as predicated or modulo-scheduled branches and counted loops (`br.cloop`) do not serve as inspection block boundaries. Following is an example of a dependency violation separated by a conditional branch:

```
     mov r4=r5
(p1) br.ret L
     mov r4=r6
```

When `p1` is false, there is a WAW dependency violation on `r4`.



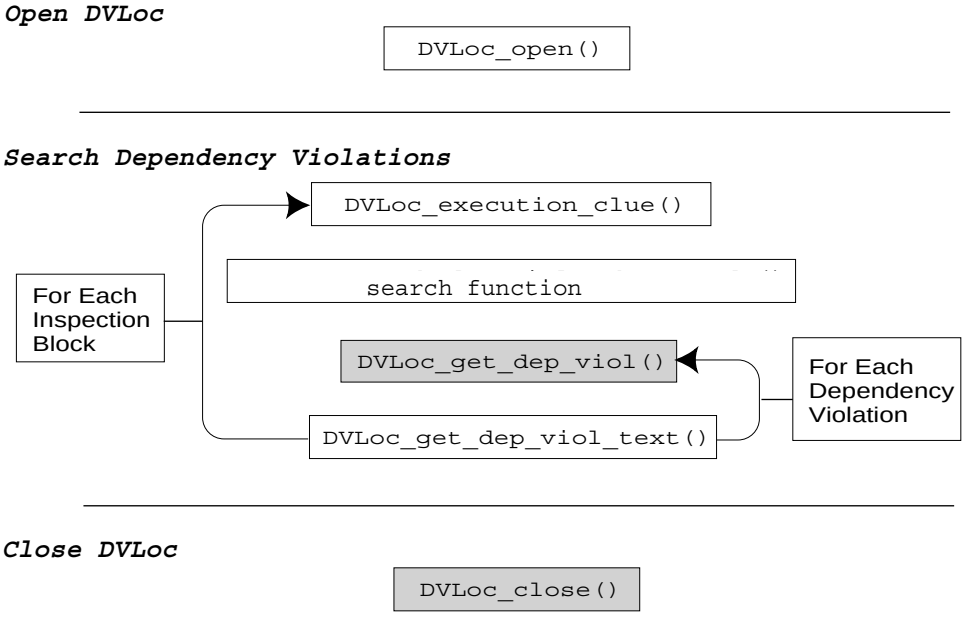
NOTE. *An entry point can be considered as a static instruction group boundary only if the code at the entry point cannot be reached by a fallthrough path.*

Close DVLoc

When the DVLoc services are no longer needed, free allocated memory with the function `DVLoc_close()`. Attempting to open DVLoc before it is closed causes implicit calls to `DVLoc_close()` and then `DVLoc_open()`.

Figure 2-1 shows a block diagram of the main DVLoc interfaces.

Figure 2-1 Block Diagram of the Interfaces



Static Mode

The client application must take the following steps.

Open DVLoc

Call `DVLoc_open(DVLOC_INSPECTION_MODE_STATIC)` to initialize the DVLoc structures and prepare DVLoc to receive Itanium architecture code.

Send Clues

Pass on all information about the inspection block, prior to sending the block itself, using the function `DVLoc_execution_clue()`. Transfer the information as data structures, called clues, containing the following types of information:

entry points	A clue for all the entry points in the binary code. See Glossary.
predicate registers relations	Clues containing the specific predicate relations when the client application has concrete information about some predicate relations in a specific place in the code. The information may be a result of execution of a predicate analysis, or a user annotation in the binary code.
size of rotating registers frame	A clue for every change in the size of the general registers rotating region size, to allow accurate detection of dependency violations in the CFM register.
accessed place in memory	A clue containing knowledge about the memory location accessed by `ld8.fill` and `st8.spill` instructions.

Each clue has a scope, which is the lifetime of the information given by the clue. More information about scopes and usage of clues is in Chapter 5, "Clues".

Send Block

Build the inspection block, which in this case is a block of Itanium architecture binary code. Send a pointer to that block of code to DVLoc with the function `DVLoc_search_dep_viols()`.

DVLoc searches for dependency violations in the block. The detected dependency violations are stored in a database in DVLoc. Extract the dependency violation information with the function `DVLoc_get_dep_viol()`.

DVLoc searches for dependency violations only in the current inspection block. No dependency violation detection is performed between the current inspection block and the previous block. For that reason, if some implementation of the Itanium architecture

architecture could cause two blocks of code to execute in the same cycle, then the two blocks must send the same call to DVLoc. The entry point (see Glossary) and skip dependency violation detection clues allow sending the complete binary code in a single call. DVLoc assumes that no predefined information (such as predicate relations) exists after an entry point.



NOTE. *To support future development, send as much code as possible in one call.*

Close DVLoc

When the DVLoc services are no longer needed, the client application calls the function `DVLoc_close()` to free allocated memory. Attempting to open DVLoc before it is closed causes implicit calls to `DVLoc_close()` and then `DVLoc_open()`.

Figure 2-1 shows a block diagram of the main DVLoc interfaces.

Dynamic Mode

The client application must take the following steps.

Open DVLoc

Call `DVLoc_open(DVLOC_INSPECTION_MODE_DYNAMIC)` to initialize the DVLoc structures and prepare DVLoc to receive Itanium architecture code.

Send Clues

Pass on all information about the inspection block, prior to sending the block itself, using the function `DVLoc_execution_clue()`. Transfer the information as data structures, called clues, containing the following types of information:

predicate register values — For every block, at least one clue is sent with the predicate register values at the beginning of the block. In addition, a clue must be sent for every

	change in the values of the predicate register values; for example, as a result of a compare instruction.
general register values	A clue for every `st8.spill` and `ld8.fill` instruction, and for every access to indirect register files such as `pmd`.
taken branch clue	A clue for every taken branch.
interrupt occurred	A clue for every occurrence of an interrupt or exception.

The clues are only valid for the next inspection block. After the search described in the next section, all the information sent in the clues is cleared. More information about usage of clues is in Chapter 5, "Clues".

Send Block

Build the inspection block, which in this case is a block of Itanium architecture binary code. Send a pointer to that block of code to DVLoc with the function `DVLoc_search_dep_viols_absolute_loc()`.

DVLoc searches for dependency violations in the block. The detected dependency violations are stored in a database in DVLoc. Extract the dependency violation information with the function `DVLoc_get_dep_viol()`.

The inspection block must be a set of instructions that were executed continuously, with no branching or interrupts. Dependency violations are detected between blocks sent in separate calls to `DVloc_search_dep_viol_absolute_loc()`.

Close DVLoc

When the DVLoc services are no longer needed, the client application calls the function `DVLoc_close()` to free allocated memory. Attempting to open DVLoc before it is closed causes implicit calls to `DVLoc_close()` and then `DVLoc_open()`.

Figure 2-1 shows a block diagram of the main DVLoc interfaces.

Enumerations and Constants

3

This chapter describes the DVLoc enumerations and the constants they define. Unless otherwise stipulated, the enumerations are defined in `sched_IA64.h`.

DVLoc_inspection_mode_t

DVLoc inspection mode

The type `DVLoc_inspection_mode_t` enumerates the various DVLoc modes. See Chapter 2, "Overview".

Syntax

```
typedef enum
{
    DVLOC_INSPECTION_MODE_STATIC,
    DVLOC_INSPECTION_MODE_DYNAMIC,
    DVLOC_INSPECTION_MODE_STATIC_ABSTR
} DVLoc_inspection_mode_t;
```

Constants

`DVLOC_INSPECTION_MODE_STATIC`	DVLoc in static mode.
`DVLOC_INSPECTION_MODE_DYNAMIC`	DVLoc in dynamic mode.
`DVLOC_INSPECTION_MODE_STATIC_ABSTR`	DVLoc in abstract mode.

DVLoc_inspection_block_t

DVLoc inspection block type

The type `DVLoc_inspection_block_t` enumerates the various block types supported by DVLoc. Each DVLoc model (abstract, static and dynamic) has a special block type. See Chapter 2, "Overview".

Syntax

```
typedef enum
{
    DVLOC_INSPECT_INSTR_GROUP,
    DVLOC_INSPECT_FUNCTION,
    DVLOC_INSPECT_INSTR_STREAM
} DVLoc_inspection_block_t;
```

Constants

`DVLOC_INSPECT_INSTR_GROUP`	Block type in abstract mode.
`DVLOC_INSPECT_FUNCTION`	Block type in static mode.
`DVLOC_INSPECT_INSTR_STREAM`	Block type in dynamic mode.

DVLoc_env_type_t

DVLoc environment variables

The type `DVLoc_env_type_t` enumerates the various environment settings of DVLoc. The only usage for this type is through the `DVLoc_set_env()` and `DVLoc_get_env()` functions.

Syntax

```
typedef enum
{
    DVLOC_ENV_OPT_FLAGS,
    DVLOC_ENV_LAST
} DVLoc_env_type_t;
```

Constants

`DVLOC_ENV_OPT_FLAGS`	Relevant only in dynamic mode. Selects depth of detection. See Chapter 6, "Functions".
`DVLOC_ENV_LAST`	Not in use.

Instr_id_t

Intel® Itanium™ instructions

The type `inst_id_t` is defined in `sched_enums.h`. It enumerates the instructions of the Itanium architecture, represented by abstract instructions.

IA64_Resource_t

Itanium resources

The type `IA64_Resource_t` is defined in `sched_enums.h`. It enumerates the Itanium architecture resources.

EM_Decoder_Reg_Name

Register operands

The type `EM_Decoder_Reg_Name` is defined in `sched_enums.h`. It defines the register operands for abstract instructions.

DVLoc_dep_viol_type_t

Dependency violations

The type `DVLoc_dep_viol_type_t` enumerates the supported types of dependency violations.

Syntax

```
typedef enum
{
   DVLOC_DEPVIOL_TYPE_RESDEP,
   DVLOC_DEPVIOL_TYPE_PIIG,
} DVLoc_dep_viol_type_t;
```

Constants

`DVLOC_DEPVIOL_TYPE_RESDEP`	A resource data dependency violation.
`DVLOC_DEPVIOL_TYPE_PIIG`	A Piig violation.

DVLoc_dep_viol_id_t

Dependency violation IDs

The type `DVLoc_dep_viol_id_t` enumerates the various dependency violation IDs.

Syntax

```
typedef enum
{
   DVLOC_DEPVIOL_NONE,
   DVLOC_DEPVIOL_RAW_RESDEP,
   DVLOC_DEPVIOL_WAW_RESDEP,
   DVLOC_DEPVIOL_WAR_RESDEP,
   DVLOC_DEPVIOL_INSTR_NOT_FIRST,
   DVLOC_DEPVIOL_INSTR_NOT_LAST,
} DVLoc_dep_viol_id_t;
```

Constants When ID is DVLOC_DEPVIOL_TYPE_RESDEP

`DVLOC_DEPVIOL_RAW_RESDEP`	Read After Write resource dependency violation
`DVLOC_DEPVIOL_WAW_RESDEP`	Write After Write resource dependency violation
`DVLOC_DEPVIOL_WAR_RESDEP`	Write After Read resource dependency violation

Constants When ID is DVLOC_DEPVIOL_TYPE_PIIG

`DVLOC_DEPVIOL_INSTR_NOT_FIRST`	Instruction should be first in the instruction group and is not.
`DVLOC_DEPVIOL_INSTR_NOT_LAST`	Instruction shoud be last in the instruction group and is not.

DVLoc_err_t

Return codes

The type `DVLoc_err_t` enumerates the various values returned by DVLoc functions.

Syntax

```
typedef enum
{
   DVLOC_ERR_NOERR,
   DVLOC_ERR_MEM_EXHAUST,
   DVLOC_ERR_DECODER_ERROR
   DVLOC_ERR_CLUE_CONFLICT_W_BIN,
```

```
    DVLOC_ERR_CLUE_CONFLICT_W_CLUE,
    DVLOC_ERR_INVALID_CLUE,
    DVLOC_ERR_OUT_OF_RANGE,
    DVLOC_ERR_INVALID_PARAM,
    DVLOC_ERR_CLOSED,
    DVLOC_ERR_INTERNAL,
    DVLOC_ERR_WRONG_OPERANDS,
    DVLOC_ERR_PREDICATE_MUST_BE_ZERO,
    DVLOC_ERR_SRC5_MISSING,
    DVLOC_ERR_INVALID_FLAGS,
    DVLOC_ERR_INVALID_INSTRUCTION_ID,
    DVLOC_ERR_ERROR
} DVLoc_err_t;
```

Constants

`DVLOC_ERR_NOERROR`	The function completed successfully.
`DVLOC_ERR_MEM_EXHAUST`	No memory is available. This is a fatal error. Close DVLoc, free some memory, and open DVLoc again.
`DVLOC_ERR_DECODER_ERROR`	DVLoc's internal decoder failed, probably because the binary code sent for analysis contained illegal Itanium architecture code.
`DVLOC_ERR_INVALID_CLUE`	Clue contains invalid data.
`DVLOC_ERR_OUT_OF_RANGE`	The dependency violation number accessed is beyond the total detected number of dependency violations. See "DVLoc_get_dep_viol()" in Chapter 6.
`DVLOC_ERR_INVALID_PARAM`	An invalid parameter passed to a function.
`DVLOC_ERR_CLOSED`	DVLoc is closed. No services are available.
`DVLOC_ERR_INTERNAL`	DVLoc internal error.
`DVLOC_ERR_WRONG_OPERANDS`	Encoded operands in the `AbstrInstrInfo` structure do not match the instruction ID.
`DVLOC_ERR_PREDICATE_MUST_BE_ZERO`	Instructions with the given ID cannot have a qualifying predicate.

DVLOC_ERR_SRC5_MISSING	The *src5_reg* field must contain a value of a general register.
DVLOC_ERR_INVALID_FLAGS	The *em_flags* field contains an illegal value.
DVLOC_ERR_INVALID_INSTRUCTION_ID	The *inst* field in the AbstrInstrInfo structure contains a number that is bigger than the possible IDs.
DVLOC_ERR_ERROR	General error.

DVLoc_certainty_t

Certain and potential violations

The type DVLoc_certainty_t distinguishes between certain and potential dependency violations. See Glossary.

Syntax

```
typedef enum
{
   DVLOC_CERTAINTY_CERTAIN,
   DVLOC_CERTAINTY_POTENTIAL,
} DVLoc_certainty_t;
```

Constants

DVLOC_CERTAINTY_CERTAIN	Definite dependency violations.
DVLOC_CERTAINTY_POTENTIAL	Possible dependency violations.

Dependency_semantics_t

Semantics of the dependency violation

The type `Dependency_semantics_t` distinguishes between different semantics of dependency violations. In the Itanium architecture there are four types of dependency violations semantics. The semantics indicates the difference in the effect the dependency violations have and the means required to solve them:

`ImpliedF`	The most common form of dependency violation semantic. This is the only semantic that can occur in non-priviliged code. And this is the only one that will be detected, unless using DVLoc in dynamic mode in Complete Dependency Semantics mode. See "DVLoc_setenv()" in Chapter 6. A dependency violation with the `ImpliedF` semantic returns an undefined value, and an Illegal Operation fault may be taken. To solve a dependency violation, the pair of instructions involved in the violation, must be in separate instruction groups.
`data,implied,instr`	See the *Intel® Itanium™ Architecture Software Developer's Manual* for information about data, instruction, and implied serialization semantics.

Syntax

```
typedef enum
{
   DEPENDENCY_SEMANTICS_NONE     ,
   DEPENDENCY_SEMANTICS_IMPLIEDF ,
   DEPENDENCY_SEMANTICS_DATA     ,
   DEPENDENCY_SEMANTICS_INSTR    ,
   DEPENDENCY_SEMANTICS_IMPLIED
} Dependency_semantics_t;
```

Constants

`DEPENDENCY_SEMANTICS_IMPLIEDFImpliedF` dependency violation semantics.

`DEPENDENCY_SEMANTICS_DATA` Data dependency violation semantics.

`DEPENDENCY_SEMANTICS_INSTR` Instruction dependency violation semantics.

`DEPENDENCY_SEMANTICS_IMPLIED` Implied dependency violation semantics.

DVLoc_clue_type_t

Types of clues

The type `DVLoc_clue_type_t` enumerates the types of clues that the client application must supply to DVLoc.

Syntax

```
typedef enum
{
    DVLOC_CLUE_PRED_GROUP_REL,
    DVLOC_CLUE_PRED_SAC_SET,
    DVLOC_CLUE_PRED_SAC_SET_ALL,
    DVLOC_CLUE_PRED_SAC_ADD,
    DVLOC_CLUE_PRED_SAC_RESET,
    DVLOC_CLUE_SINGLE_REG,
    DVLOC_CLUE_PRED_PAIR_REL,
    DVLOC_CLUE_PRED_GROUP_REL,
    DVLOC_CLUE_PRED_VECTOR,
    DVLOC_CLUE_EXTERN_ENTRY,
    DVLOC_CLUE_BRANCH_TAKEN,
    DVLOC_CLUE_SKIP_DEPVIOL_DETECTION
    DVLOC_CLUE_LABEL,
    DVLOC_CLUE_INTERRUPT_OCCURED,
    DVLOC_CLUE_ROT_GREG_REGION_SIZE,
    DVLOC_CLUE_MEM_BASE_OFFSET,
    DVLOC_CLUE_PRED_VECTOR
    DVLOC_CLUE_PRED_REL_ARRAY,
    DVLOC_CLUE_LAST
} DVLoc_clue_type_t;
```

Constants

DVLOC_CLUE_PRED_GROUP_REL	Annotates a relation between a group of predicates. For example, P1, P2, and P3 are mutex.
DVLOC_CLUE_PRED_SAC_SET	Relevant for static and abstract modes. The relation between a set of predicates, preserved during function calls.
DVLOC_CLUE_PRED_SAC_SET_ALL	Relevant for static and abstract modes. The relations between all the predicates, preserved during function calls.
DVLOC_CLUE_PRED_SAC_ADD	Relevant for static and abstract modes. Adds a set of predicates to the set of the safe across call (SAC) predicates.
DVLOC_CLUE_PRED_SAC_RESET	Relevant for static and abstract modes. The end of the scope of all the other SAC clues. From this point, none of the predicates is safe across calls.
DVLOC_CLUE_SINGLE_REG	For all modes. Client application uses it when passing the value of a general register to DVLoc.
DVLOC_CLUE_PRED_PAIR_REL	Relevant for abstract and static modes. The relation of two predicate registers at a given point. Only for imply relations.
DVLOC_CLUE_PRED_GROUP_REL	Relevant for abstract and static modes. The relations between groups of predicate registers at a given point. Only for implication and clear relations.
DVLOC_CLUE_PRED_VECTOR	Relevant only in dynamic mode. The value of the predicate registers at a given point.
DVLOC_CLUE_EXTERN_ENTRY	Relevant only for static mode. A place to which a branch might jump.
DVLOC_CLUE_BRANCH_TAKEN	Relevant only for dynamic mode. Indicates if a branch was taken.
DVLOC_CLUE_SKIP_DEPVIOL_DETECTION	Only for static mode. Informs DVLoc to ignore part of the binary code.

DVLOC_CLUE_LABEL	Relevant only for static mode. Indicates that a location can be the destination of a branch only within the binary code sent in the subsequent call to the function DVLoc_search_dep_viols().
DVLOC_CLUE_INTERRUPT_OCCURED	Relevant only for dynamic mode. Indicates that an interrupt occurred before (or in) the next block.
DVLOC_CLUE_ROT_GREG_REGION_SIZE	Relevant for all modes. The rotating registers' rotating region size.
DVLOC_CLUE_PRED_MEM_BASE_OFFSET	Only relevant for abstract and static modes. Indicates the memory locations of store and load operations of ld8.fill and st8.spill instructions.
DVLOC_CLUE_PRED_VECTOR	Only for dynamic mode. Used when passing the values of the predicate registers.
DVLOC_CLUE_PRED_REL_ARRAY	Only relevant for abstract and static mode. Transfers a group of relations between predicate registers.
DVLOC_CLUE_LAST	Not in use.

For further explanations of the clues, see Chapter 5, "Clues".

DVLoc_clue_timing_t

Clue timing

The type DVLoc_clue_timing_t enumerates the various timing for a clue. A clue can take effect before or after the instruction to which it is attached.

Syntax

```
typedef enum
{
   DVLOC_CLUE_BEFORE_EXECUTION,
   DVLOC_CLUE_AFTER_EXECUTION
} DVLoc_clue_timing_t;
```

Constants

`DVLOC_CLUE_BEFORE_EXECUTION`	The clue takes effect before the instruction to which it is attached.
`DVLOC_CLUE_AFTER_EXECUTION`	The clue takes effect after the instruction to which it is attached.

DVLoc_pred_pair_rel_t

Pairs of predicate relationships

The type `DVLoc_pred_pair_rel_t` enumerates the relationships between a pair or pairs of predicate registers.

Syntax

```
typedef enum
{
    DVLOC_PRED_PAIR_REL_KEEP,
    DVLOC_PRED_PAIR_REL_UNKNOWN,
    DVLOC_PRED_PAIR_REL_MUTEX,
    DVLOC_PRED_PAIR_REL_IMPLIES
} DVLoc_pred_pair_rel_t;
```

Constants

`KEEP`	Do not change any existing predicate relations established for the given pair.
`UNKNOWN`	The relationship between a given pair of predicate registers is not known.

`MUTEX`	Two predicate registers cannot be true simultaneously, but one can be true when the other is false. Use this form only with the data type `PredInfoArray` and not in the data structure `DVLoc_pred_pair_t`. See Chapter 5, "Clues".
`IMPLIES`	If the first predicate register is true, then the second predicate register is true.



NOTE. *The implication relation* `IMPLIES` *is an ordered relation. Unlike the mutex relation, the order in which the predicates in the relations are sent is important.*

DVLoc_pred_group_rel_t

A group of predicate relationships

The type `DVLoc_pred_group_rel_t` enumerates the relationships within a group of predicate registers.

Syntax

```
typedef enum
{
    DVLOC_PRED_GROUP_REL_CLEAR,
    DVLOC_PRED_GROUP_REL_MUTEX
} DVLoc_pred_group_rel_t;
```

Constants

`CLEAR`	All the relations are cleared that refer to one or more predicate in the group. The value of the predicates in the group may change.
`MUTEX`	The predicate registers in the group cannot be true simultaneously, but one can be true when any other is false.

Data Structures 4

This chapter describes the data structures DVLoc uses.

Uint64

Represents a native 64-bit unsigned integer

When using DVLoc on Windows NT*, the type `Uint64` is `unsigned __int64` (the default 64-bit unsigned integer).

When using DVLoc on UNIX*, the type `Uint64` is `unsigned long long` (the default 64-bit unsigned integer in UNIX).

DVLoc_res_info_t

Contains information about a resource

```
typedef struct
{
    IA64_Resource_t  res;
    const char*      name;
} DVLoc_res_info_t;
```

Parameters

res	A resource enumeration.

name	A text description of the resource.

DVLoc_dep_viol_t

Contains info from DVLoc_get_dep_viol()

```
typedef struct
{
  Uint64  instr_loc;
  unsigned  abstr_instr_loc;
  DVLoc_dep_viol_id_t  id;
  DVLoc_certainty_t  certainty;
  DVLoc_dep_viol_type_t  data_type;
  Dependency_semantics_t  dep_semantics;
  union {
    struct res_dependency_s
    {
      DVLoc_res_info_t  res_info; /* Resource involved in the D.V.*/
      Uint64  instr_loc;
      unsigned  abstr_instr_loc;
    } res_dependency;
  } data;
} DVLoc_dep_viol_t;
```

Parameters

instr_loc	Relevant only in static and dynamic modes. The 64-bit address of the first instruction in the pair of instructions that caused the resource dependency violation, or the location of the instruction that is positioned incorrectly in the instruction group.
abstr_instr_loc	Relevant only for abstract mode. The location of the second instruction in the pair of instructions that caused the resource dependency violation.
id	Dependency violation ID. See "DVLoc_dep_viol_id_t" in Chapter 3.

certainty	Dependency violation certainty, whether certain or potential. See "DVLoc_certainty_t" in Chapter 3.
dep_semantics	Relevant only in dynamic mode. Determines the type of the dependency violation: `impliedF`, `data`, `instr`, or `implied`.
data_type	Type of data; dictates how to read the union data. For `DVLOC_DEPVIOL_TYPE_RESDEP`, the field `res_dependency` in the `data` union contains the following valid information:
res_info	Data type containing information about the resource on which the dependency violation occurred.
instr_loc	Relevant only for static and dynamic modes. The 64-bit address of the second instruction in the pair of instructions that caused the resource dependency violation.
abstr_instr_loc	Relevant only for abstract mode. The location of the second instruction in the pair of instructions that caused the resource dependency violation.



NOTE. *When the field* `id` *equals* `DVLOC_DEPVIOL_TYPE_PIIG`, *the* `data_type` *field and the structure data are irrelevant.*

AbstrInstrInfo - Abstract Instruction

Describes an Intel® Itanium™ architecture instruction for abstract mode

```
struct DynDepViolDecInfo {
    EM_Decoder_Inst_Id       inst;
    long                     pred_nb;
    EM_Decoder_Reg_Name      src1_reg;
    EM_Decoder_Reg_Name      src2_reg;
    EM_Decoder_Reg_Name      src3_reg;
    EM_Decoder_Reg_Name      src4_reg;
```

```
    EM_Decoder_Reg_Name      src5_reg;
    EM_Decoder_Reg_Name      dst1_reg;
    EM_Decoder_Reg_Name      dst2_reg;
    unsigned int             size;
    unsigned long            em_flags;
    Uint64                   imm_data;
};
typedef struct DynDepViolDecInfo AbstrInstrInfo;
```

Parameters

inst Instruction ID.

pred_nb Index of the qualifying predicate register of the instruction. If the instruction cannot be predicated, `pred_nb` contains `0`.

dst1_reg, dst2_reg, src1_reg, src5_reg
If one of the instruction operands is one of the following groups of Itanium architecture registers, then the corresponding field in the `AbstrInstrInfo` contains the matching `EM_Decoder_Reg_Name` enumeration. The groups are general registers, floating point registers, predicate registers, control registers, application registers, and branch registers.
If the operand is a memory operand, then the client application assigns the matching `EM_Decoder_Reg_Name` of the memory base register to *src5_reg*.

size Reserved.

em_flag Indicates the stop (`;;`) after the current instruction. The client application uses the following macros when changing this field:
`set_abstr_cycle_break(x)`
where *x* is an `AbstrInstrInfo` data structure, indicating a stop after *x*.
`clear_abstr_cycle_break(x)`
where *x* is an `AbstrInstrInfo` data structure, indicating no stop after *x*.

`imm_data` If the instruction is `(qp) mov pr = r2, mask17`
then `imm_data` contains `mask`.
When encoding other instructions, this field is ignored.



NOTE. *Abstract instructions are real instructions; that is, DVLoc cannot analyze pseudo operations (pseudo-ops). The client application should replace pseudo-ops in the* `AbstrInstrInfo` *structure with information from the real instruction corresponding to the pseudo-op.*



NOTE. *See* "DVLoc_validateAbstrInstr()" in Chapter 6 *for a description of the function that validates an* `AbstrInstrInfo` *structure.Use this function to debug your encoding algorithm.*

Clues

5

Clues are features of the DVLoc API. The client application uses clues to pass information to DVLoc concerning the code about to be sent.

Sending accurate clues to DVLoc is essential for accurate dependency violation detection. Clues only affect the next inspection block, which is the first block sent to DVLoc after the clue. This is the clue's maximal scope.

For every clue, location and timing are set. See *instr_loc/abstr_instr_loc* and *clue_timing*, below.

This chapter describes the clues, which are defined in `sched_IA64.h`.

Clue Data Structure

The function `DVLoc_execution_clue()` sends the DVLoc structure `DVLoc_clue_t`. Immediately following is the outer data structure. The inner data structures such as `DVLoc_pred_pair_t` are described later in this chapter, ordered by functionality. Inner enumerations such as `DVLOC_clue_timing_t` are described in Chapter 3, "Enumerations and Constants".

DVLoc_clue_t

Sent with the function DVLoc_execution_clue()

This structure is the basis for all the clues described in this chapter.

Syntax

```
typedef struct
{
  Uint64  instr_loc;
  unsigned  abstr_instr_loc;
  DVLoc_clue_timing_t  clue_timing;
  DVLoc_clue_type_t  data_type;

  union  data {
    DVLoc_dpcat_vector_mask_t  safe_across_calls_mask;
    DVLoc_pred_group_t  pred_group;
    DVLoc_pred_pair_t  pred_pair;
    DVLoc_pdcat_vector_value_t  pred_vector;
    unsigned long  skip_dep_viol_detection_nb_bytes;
    unsigned  gr_rotating_region_size;
    PredInfoArray  rel_arr;
    struct {
        unsigned  offset;
        unsigned  region_index;
    } mem_base_offset;
  } data;
} DVLoc_clue_t;
```

Parameters

instr_loc	Relevant only for static and dynamic modes. This field indicates the clue location, which is a 64-bit address. The upper 60 bits indicate the bundle address, and the lower 4 bits indicate the slot number.
abstr_instr_loc	Relevant only for abstract mode. This field indicates the clue location. The location is an index to an instruction in the inspection block.
clue_timing	Determines when the clue takes effect. See "DVLoc_clue_timing_t" in Chapter 3.
data_type	The type of the clue. It indicates which field is filled in the `data` union. See "DVLoc_clue_type_t" in Chapter 3.
data	The actual clue data. For each clue, the appropriate method of using the `data` field is described.

Predicate Relations Clues

The predicate relations clues described in this section are relevant only for abstract and static modes.

In abstract mode, the client application sends clues indicating all the predicate relations that exist in the beginning of the block with `abstr_instr_loc = 0`. In addition, for every location of the code at which predicate registers relations change, the client application must send a clue. The scope of the predicate relation clues is until the end of the inspection block.

In static mode, DVLoc applies limited analysis to deduce predicate relationships from the code. Not all the existing predicate relationships are deduced automatically. For example, DVLoc avoids conditional analysis. In addition, when a location is an entry point, DVLoc cannot assume anything about the predicate relations after that location, so clues must be sent with any information the client application already has.

Clues can represent three types of predicate register relations:

mutex	When two or more predicate registers are mutex in a given region of code, it means that in that region only one of them is true.
implication	When predicate register Pi implies Pj at a given location, it means that if, at that location, Pi is true, then at that location Pj is also true. When the Pi is false, no assumptions are made about Pj.
clear	Clear Pi means to clear all previous relations involving Pi.

The influence of these types of predicate register relations is explained below in conjunction with the clues.

There are two approaches to sending predicate relation clues to DVLoc:

single relation	send single relation between a pair or a group of predicates
set of relations	send a set of relations all at once

Both approaches can be used simultaneously.



NOTE. *The predicate relations examples are for abstract mode. In static mode, DVLoc deduces the specific predicate relations in these examples automatically from the code.*

Single Relation Approach

In the single relation approach, the client application sends every relation between a pair or a group of predicate registers as a separate clue. The advantage of this approach is its ease of use. The drawback is the large number of function calls required when the client application sends many relations for a single inspection block.

The value of `data_type` in the clue data structure `DVLoc_clue_t` must be `DVLOC_CLUE_PRED_PAIR_REL` or `DVLOC_CLUE_PRED_GROUP_REL`.

The field filled in the `data` union is `pred_pair` or `pred_group`.

DVLoc_pred_pair_t

Passes predicate register pairs relations

The `DVLoc_pred_pair_t` clue sends information regarding pairs of predicate registers.

Syntax

```
typedef struct
{
  DVLoc_pred_pair_rel_t pred_info;
  int pred_nb_1;
  int pred_nb_2;
} DVLoc_pred_pair_t;
```

Parameters

pred_info	The relation of the two predicate registers. The only valid value is `imply`.
pred_nb_1, *pred_nb_2*	The indexes of the two predicate registers. P0 is `0`, P1 is `1`, and so on. For mutex, *pred_nb_1* and *pred_nb_2* are mutex. For implication, *pred_nb_1* implies *pred_nb_2*.

DVLoc_pred_group_t

Passes predicate register group relations

The `DVLoc_pred_group_t` clue sends information regarding groups of predicate registers.

Syntax

```
typedef struct
{
  DVLoc_pred_group_rel_t rel_type;
  DVLoc_pdcat_vector_mask_t pred_mask_vector;
} DVLoc_pred_group_t;
```

Parameters

rel_type	The relation of the group of predicate registers. The only valid values are `mutex` and `clear`.
pred_mask_vector	A 64-bit mask indicating the predicates in the group. The indexes of the two predicate registers, where P0 is `0`, P1 is `1`, and so on.

Example 1

Suppose that the following code is sent to DVLoc:

```
     cmp.eq p1,p2=r1,r2;;
(p1) mov r4=r5
(p2) mov r4=r6;;
```

P1 and P2 are mutex after the compare instruction, so the client application must send a clue to DVLoc, similar to the following code:

```
DVLoc_clue_t a_clue;

a_clue.abstr_instr_loc = 0;
a_clue.data_type = DVLOC_CLUE_PRED_GROUP_REL;
a_clue.clue_timing = DVLOC_CLUE_AFTER_EXECUTION;
a_clue.data.pred_group.rel_type = DVLOC_PRED_GROUP_REL_MUTEX;
a_clue.data.pred_group.pred_mask_vector = 6;
DVLoc_execution_clue(&a_clue);
```

Example 2

The following code is sent to DVLoc:

```
     cmp.eq p1,p2=r1,r2;;
(p1) mov ar.ec = r4
     cmp p2,p3 = r5,r6
(p2) br.call.dpnt L
```

Two clues must be sent, declaring the new relations formed as a result of the two compare instructions:

1. the location of the first compare instruction indicating the mutex relation between P1 and P2
2. the location of the second compare instruction indicating the mutex relation between P2 and P3

However, notice that the second compare instruction destroys the mutex relation between P1 and P2. Thus, a third clue should be sent:

3. location of the second compare, setting the relation between P1 and P2 to `UNKNOWN`



NOTE. *To clear the relations between one predicate register and all the others, you can use a single clue: DVLoc_pred_group_rel_clear.*

Example 3

The client application sends the following code to DVLoc:

```
(p1) cmp.eq.unc p2=r1,r2;;
(p2) mov r4=r5
(p1) br.call.dpnt L
       mov r4=r5
```

The compare instruction creates an implication relation P2->P1. A clue indicating this relation is sent with the location of the compare, including the *clue_timing* field DVLOC_CLUE_TIMING_AFTER_EXECUTION.

Example 4

Consider this example:

```
(p2) cmp.eq.unc p1=r12,r13;;
(p1) mov r1=r11
(p2) br L
       mov r1=r14
```

The first instruction establishes the relation P1->P2, as the unconditional compare can set P1 to true only if P2 is also true. The client application sends a clue to DVLoc as follows:

```
DVLoc_clue_t a_clue;

a_clue.abstr_instr_loc = 0;
a_clue.data_type = DVLOC_CLUE_PRED_PAIR_REL;
a_clue.clue_timing = DVLOC_CLUE_AFTER_EXECUTION;
a_clue.data.pred_pair.pr_nb_1 = 1;
a_clue.data.pred_pair.pr_nb_2 = 2;
a_clue.data.pred_pair.pred_info = DVLOC_CLUE_PRED_PAIR_IMPLY;
DVLoc_execution_clue(&a_clue);
```

Example 5

```
(p2) cmp.eq.unc p1=r12,r13;;    // p1->p2
(p1) mov r1=r11
     cmp.eq  p2=r15,16          // p1->p2 is no longer valid
(p2) br L
       mov r1=r14
```

This example looks almost the same as the previous one. The difference is the compare that writes to P2 before the branch. After this write, it is not clear if the relation exists, as even if P2 was true at the first instruction, it may not be true after the compare. In this case, DVLoc reports a WAW dependency violation on R1.

Set of Relations Approach

The set of relations approach allows one clue to send complete sets of predicate relations. The advantage of this approach is that one function call allows the client application to send many relations for the one inspection block. However, it is more complicated to use.

The value of the field *data_type* in the clue data structure `DVLoc_clue_t` must be `DVLOC_CLUE_PRED_REL_ARRAY`. *rel_arr* is the corresponding field in the data union.

The three following data types are nested: `PredInfoArray` uses `PredInfoArrayEntry`, which uses `DVLoc_pdcat_relation_vector_t`.

PredInfoArray

Implements an array containing PredInfoArrayEntry entries

Syntax

```
typedef struct
{
   unsigned  size;
   PredInfoArrayEntry*  arr;
   unsigned  flags;
} PredInfoArray;
```

Parameters

size	The size of the array pointed at by `arr`.
arr	An array of predicate registers relations.

flags	PRED_INFO_FLAG_STRONG tells DVLoc to set all the predicate relations to DVLoc_PRED_PAIR_REL_UNKNOWN before applying clue information. PRED_INFO_FLAG_WEAK tells DVLoc to add the information in the clue to the information gathered prior to this clue location.

PredInfoArrayEntry

Describes relations between a predicate register and all other predicate registers

The PredInfoArrayEntry structure describes relations between a predicate register with index *pr_nb* and all the other predicate registers.

When the relation in the *pred_info* vector is mutex, it means that for some registers there is a mutex relation between *pr_nb* and the given register.

When the relation is implication, it means that if *pr_nb* is true, then the predicate register is also true.

Syntax

```
typedef struct
{
   unsigned pr_nb;
   DVLoc_pdcat_relation_vector_t pred_info;
} PredInfoArrayEntry;
```

Parameters

pr_nb	Contains the index of the predicate register.
pred_info	Contains all the relations between predicate registers.

DVLoc_pdcat_relation_vector_t

Contains 64 predicate register relations, set with macro `SET_PRED_PAIR_REL_INFO`

The `DVLoc_pdcat_relation_vector_t` structure holds 64 predicate relations. The following describes the use of the macro, which is the recommended method for handling this structure.

Syntax

`SET_PRED_PAIR_REL_INFO(`*`pdcat_rel_vector`*`, `*`pr_nb`*`, `*`pred_info`*`)`

Parameters

`pdcat_rel_vector`	A variable of type `DVLoc_pdcat_relation_vector_t`.
`pr_nb`	The index of the predicate register to which the client application adds the information.
`pred_info`	One of the enumerations in `DVLoc_pred_pair_rel_t`. See "DVLoc_pred_pair_rel_t" in Chapter 3.

Example

Assume the client application knows at a certain location that P1, P2, P3, and P4 are all mutex, and no other predicate relations are known at this location. The following code example shows how to pass this information to DVLoc.

```
DVLoc_clue_t a_clue;

unsigned clue_loc = 0;
PredInfoArrayEntry relArr[64]; // The predicates relations array
DVLoc_pred_pair_rel_t mutex=DVLOC_PRED_PAIR_REL_MUTEX;

a_clue.data_type = DVLOC_CLUE_PRED_REL_ARRAY;
a_clue.clue_timing = DVLOC_CLUE_BEFORE_EXECUTION;

a_clue.abstr_instr_loc = 0;

// three relations are sent
a_clue.data.rel_arr.size = 3;
```

```
// This clue contains the only predicate relations that are known
// at this location
a_clue.data.rel_arr.flags = PRED_INFO_FLAG_STRONG;
// Set pointer to the relations array
a_clue.data.rel_arr.arr = relArr;
// Set information in the relations array
// relations of pr1
relArr[0].pr_nb = 1;
// reset the pdcat relation info
RESET_PRED_PAIR_REL_INFO(relArr[0].pred_info);
// p1 and p2 are mutex
SET_PRED_PAIR_REL_INFO(relArr[0].pred_info,2,mutex);
// p1 and p3 are mutex
SET_PRED_PAIR_REL_INFO(relArr[0].pred_info,3,mutex);
// p1 and p4 are mutex
SET_PRED_PAIR_REL_INFO(relArr[0].pred_info,4,mutex);
// relations of pr2
relArr[1].pr_nb = 2;
// reset the pdcat relation info
RESET_PRED_PAIR_REL_INFO(relArr[1].pred_info);
// p1 and p2 are mutex
SET_PRED_PAIR_REL_INFO(relArr[1].pred_info,1,mutex);
// p2 and p3 are mutex
SET_PRED_PAIR_REL_INFO(relArr[1].pred_info,3,mutex);
// p2 and p4 are mutex
SET_PRED_PAIR_REL_INFO(relArr[1].pred_info,4,mutex);
// relations of pr3
relArr[2].pr_nb = 3;
// reset the pdcat relation info
RESET_PRED_PAIR_REL_INFO(relArr[2].pred_info);
// p3 and p1 are mutex
SET_PRED_PAIR_REL_INFO(relArr[2].pred_info,1,mutex);
// p3 and p2 are mutex
SET_PRED_PAIR_REL_INFO(relArr[2].pred_info,2,mutex);
// p3 and p4 are mutex
SET_PRED_PAIR_REL_INFO(relArr[2].pred_info,4,mutex);
```

```
// send the clue
DVLoc_execution_clue(&a_clue);
```



CAUTION. *The predicate relations array clue allows the client application to send contradictory information. It is the client application's responsibility to ensure this does not happen. For example, an array can be sent with two vectors, one with the relations of P1, and the other with the relations of P2. In the first vector the client application can set the relation of P1 with P2 to mutex. In the second vector, the client application can set the relation between P2 and P1 to unknown. This contradiction might reduce the accuracy of the dependency violation detection.*

Safe Across Calls Clues

Passes SAC clues

The safe across calls (SAC) clues instruct DVLoc to preserve some or all of the predicates relations after a branch call `br.call`.

`clue_timing` is set to `DVLOC_CLUE_BEFORE_EXECUTION`.

The values for *`data_type`* are:

`DVLOC_CLUE_PRED_SAC_SET`	Some predicates are SAC. *`data`* is important.
`DVLOC_CLUE_PRED_SAC_SET_ALL`	All predicates are SAC. Values for *`data`* are unimportant.
`DVLOC_CLUE_PRED_SAC_ADD`	From this point, additional predicates are SAC. *`data`* is important.
`DVLOC_CLUE_PRED_SAC_RESET`	None of the predicates is SAC. Values for *`data`* are unimportant.

The scope of the clues is until the end of the inspection block, or the until the next reset.

The data of the clue is:

```
DVLoc_pdcat_vector_mask_t  safe_across_calls_mask;
```

Parameter

safe_across_calls_mask	A 64-bit mask indicating the SAC predicates.

Rotating Region Size Clue

The rotating register size clue allows the client application to pass the general registers rotating region size at a location of the inspection block to DVLoc. DVLoc uses the size to determine if a general register is a rotating register. This clue must be sent for at least every occurrence of the `alloc` instruction.

The rotating registers identification is necessary because accessing a rotating register implies a read of the CFM register.

The scope of the clue continues until another rotating general register size clue is sent.

By default, DVLoc assumes that the size of the general register rotating region is `0`. If this is not so, the client application should send a clue at the beginning of the inspection block with index `0` as the location.

Parameters

clue_timing and location	See "Clue Data Structure".
data_type	DVLOC_CLUE_ROT_GREG_REGION_SIZE
data.gr_rotating_region_size	The new rotating registers region size.

Example

Assume that the first instruction in the inspection block changes the general registers rotating region size to `8`. The client application includes the following code:

```
DVLoc_clue_t a_clue;
```

```
a_clue.abstr_instr_loc = 0;            // first instruction location
a_clue.data_type = DVLOC_CLUE_ROT_GREG_REGION_SIZE;
a_clue.data.gr_rotating_region_size = 8; // the new size
a_clue.clue_timing = DVLOC_CLUE_AFTER_EXECUTION;
DVLoc_execution_clue(&a_clue);
```

st8.spill and ld8.fill Clue

The `st8.spill` and `st8.fill` clue is relevant only for static and abstract modes.

AR[UNAT] is an application register in the Intel® Itanium™ architecture, described in the *Intel® Itanium™ Architecture Software Developer's Manual*.

An `st8.spill` instruction writes bit `X` of AR[UNAT], where `X` equals the value in bits 8:3 of the store's data address. An `ld8.fill` instruction reads bit `Y` of AR[UNAT], where `Y` equals the value in bits 8:3 of the load's data address. These write and read instructions can cause RAW and WAW dependency violations on AR[UNAT] bits. In order to eliminate false dependency violation reports when several accesses to the AR[UNAT] bits are done in the same instruction group, the client application must hint the address of the accessed memory to DVLoc.

By default, DVLoc assumes that every `st8.spill` and `ld8.spill` writes and reads respectively all the `unat` bits.

The field filled in the `DVLoc_clue_t` structure is *`mem_base_offset`*.

`clue_timing` is set to `DVLOC_CLUE_BEFORE_EXECUTION`, and *`data_type`* to `DVLOC_CLUE_MEM_BASE_OFFSET`.

Syntax

See "DVLoc_clue_t".

Parameters

`data.mem_base_offset.offset`	The offset of the accessed memory, relative to the beginning of the scratch memory region.
`data.mem_base_offset.region_index` `0`	

Example

Consider the following example:

```
(0) movl r2=MY_SCRATCH;;
(1) mov  r4=r2
(2) add  r5=r2,8
(3) add  r6=r2,16;;
```

At this point we know that `r5=r4+8`, and `r6=r5+8`

```
(4) st8.spill [r4]=r32
(5) st8.spill [r5]=r33
(6) ld8.fill  r34=[r6];;
```

The instructions `4-6` above access three different bits of the UNAT resource, thus DVLoc should not report any WAW or RAW dependency violations. To eliminate a false dependency violation report, the client application must send the following clues:

- For instruction `(4)` send a clue with `offset = 0`.
- For instruction `(5)` send a clue with `offset = 8`.
- For instruction `(6)` send a clue with `offset = 16`.

The following code shows one method of generating the second clue:

```
DVLoc_clue_t a_clue;

a_clue.data_type = DVLOC_CLUE_MEM_BASE_OFFSET;
a_clue.timing = DVLOC_CLUE_BEFORE_EXECUTION;
a_clue.data.mem_base_offset.offset = 8;
a_clue.data.mem_base_offset.region_index = 0;

DVLoc_execution_clue(&a_clue);
```

Register Values Clue

The register values clue is only relevant for dynamic mode.

In the following cases, DVLoc requires the value of a general register in order to perform accurate dependency violations detection:

- when using the `st8.spill` and `ld8.fill` instructions, the value of the base register is required

- when accessing an indirect register file such as `pmd`, the value of the index general registers is required

If the client application does not send a clue, DVLoc might falsely report a dependency violation.

The scope of this clue is one instruction.

`clue_timing` is set to `DVLOC_CLUE_BEFORE_EXECUTION`, and *`data_type`* to `DVLOC_CLUE_SINGLE_REG`.

Syntax

```
typedef struct
{
    EM_Decoder_Reg_Name  name;
    Uint64          content;
} DVLoc_single_reg_t;
```

Parameters

`data.single_reg.name`	The `EM_Decoder_Reg_Name` enumeration representing the general register, whether base memory or index. Only general registers are supported.
`data.single_reg.content`	The value of the general register.

Example

The following code is in location 0x40.

```
{ .mfi
    addl  r5 = 0x4, r0
    nop.f  0x0
    addl  r6 = 0x5, r0 ;;
}
{ .mmi
    mov  pmd[r5] = r16    // location = 0x50
    mov  pmd[r6] = r16    // location = 0x51
    nop.i  0x0
}
```

To avoid a false dependency violation report for the writes to pmd, send two clues regarding general register values, indicating that

- the value of R5 is 4
- the value of R6 is 5

The location of the clues is the same as the location of the relevant instruction. If the client application accesses pmd again with R5 as the index register, a separate clue must be sent.

```
DVLoc_clue_t clue;
// first clue
clue.data_type = DVLOC_CLUE_SINGLE_REG;
clue.clue_timing = DVLOC_CLUE_BEFORE_EXECUTION;
clue.instr_loc = 0x50;
clue.data.single_reg.name = EM_DECODER_REG_R5;
clue.data.single_reg.content = 4;
DVLoc_execution_clue(&clue);
// second clue
clue.single_reg.name = EM_DECODER_REG_R6;
clue.single_reg.content = 5;
DVLoc_execution_clue(&clue);
```

Skip Dependency Violations Detection Clue

The skip dependency violations clue is relevant for static mode.

The code analysed by DVLoc must be legal (Itanium architecture) code. The skip dependency violations clue allows sending blocks of binary code that contain non-legal code such as data. Send the clue for every location in the binary code that contains non-legal code. DVLoc skips those areas in the binary code without trying to analyse them.

clue_timing is set to DVLOC_CLUE_BEFORE_EXECUTION, and *data_type* to DVLOC_CLUE_SKIP_DEPVIOL_DETECTION.

Parameter

data.skip_dep_viol_detection_nb_bytes	The number of bytes to skip during dependency violation detection. Value must be a multiple of 16.

Example

The following code contains data. DVLoc should skip it.

```
{ .mfi
    addl  r5 = 0x4, r0
    nop.f  0x0
    addl  r6 = 0x5, r0 ;;
}
data8 4,5,6            // 24 bytes of data + 8 bytes of alignment to 16
{ .mmi
    mov     pkr[r5] = r16  // location = 0x50
    mov     pkr[r6] = r16  // location = 0x51
    nop.i  0x0
}
```

The address of the data block is 0x16.

Examine the clue:

```
DVLoc_clue_t clue;

clue.data_type = DVLOC_CLUE_SKIP_DEPVIOL_DETECTION;
clue.clue_timing = DVLOC_CLUE_BEFORE_EXECUTION;
clue.instr_loc = 0x16;
clue.data.skip_dep_viol_detection_nb_bytes = 32;

DVLoc_execution_clue(&clue);
```



CAUTION. *Sending non-legal Itanium architecture binary code to DVLoc without a skip dependency violations clue causes DVLoc to stop the detection.*

Functions

6

This chapter describes the various functions in DVLoc's API. The functions appear in order of use, as described in Chapter 2, "Overview".

Some functions are relevant only for a specific mode. If nothing is specified, than the function can be used in all DVLoc modes.

DVLoc_open()

Opens DVLoc

```
DVLoc_err_t DVLoc_open(DVLoc_inspection_mode_t  inspection_mode)
```

Parameter

inspection_mode — Mode in which the client application inspects the input. Values:

static	DVLOC_INSPECTION_MODE_STATIC
abstract	DVLOC_INSPECTION_MODE_STATIC_ABSTR
dynamic	DVLOC_INSPECTION_MODE_DYNAMIC

Description

The DVLoc_open() function initializes the DVLoc internal databases. DVLoc must open successfully in order to call other services. DVLoc then clears the clue and dependency violation databases.

Opening DVLoc again before it is closed causes closing and clearing of the internal databases and reopening of the library.

Return Values

DVLOC_ERR_NOERROR	Function completed successfully.
DVLOC_ERR_MEM_EXHAUST	No memory is available. This is a fatal error.

DVLoc_setenv()

Sets DVLoc environment variables

```
DVLoc_err_t DVLoc_setenv(DVLoc_env_type_t env_type, long env_value);
```

Parameters

env_type	Type of the environment item to change.
env_value	New value of the environment item. Default is 0.

Description

The DVLoc_setenv() function is relevant only for dynamic mode. It allows full dependency violation detection where the detected violations are of all possible semantics. This is known as Complete Dependency Semantics mode.

The only supported value for *env_type* is DVLOC_ENV_OPT_FLAGS.

Values for *env_value* are DVLOC_ENV_FLAG_COMPLETE_DEPENDENCY_SEMANTICS and 0. When using the first value, DVLoc detects dependency violations of all the possible semantics: instr, data, implied, and ImpliedF. When the value is 0, only impliedF semantics dependency violations are detected.

Return Values

DVLOC_ERR_NOERROR	Function completed successfully.
DVLOC_ERR_INVALID_PARAM	Invalid parameter passed to function.
DVLOC_ERR_CLOSED	Library is closed; no services are available.

DVLoc_execution_clue()

Passes an execution clue to DVLoc

```
DVLoc_err_t DVLoc_execution_clue(DVLoc_clue_t*  clue)
```

Parameter

clue	A pointer to an execution clue (input).

Description

The `DVLoc_execution_clue()` function passes information in the format of clues about the next inspection block. DVLoc stores the clue in an internal database of clues, for use in detecting dependency violations. DVLoc automatically clears the clue database after searching for dependency violations in each inspection block.

Return Values

`DVLOC_ERR_NOERROR`	Function completed successfully.
`DVLOC_ERR_CLUE_CONFLICT_W_CLUE`	Two clues have a conflicting information.
`DVLOC_ERR_INVALID_CLUE`	Invalid data in clue.
`DVLOC-ERR_MEM_EXHAUST`	No memory is available. This is a fatal error.
`DVLOC_ERR-CLOSED`	Library is closed; no services are available.

DVLoc_validateAbstrInstr()

Checks that the AbstrInstrInfo info is valid

```
DVLOC_EXPIMP DVLoc_err_t DVLoc_validateAbstrInstr(
    const AbstrInstrInfo*  abs_instr)
```

Parameter

abs_instr	A pointer to an `AbstrInstrInfo` structure.

Description

The DVLoc_validateAbstrInstr() function is relevant only for abstract mode. It is a utility function, and should be activated only in debug mode. Create an array of AbstrInstInfo structures, and then call the function for each item of the array to ensure that the encoding is valid. For example, the function checks if the encoded operands match the operand types corresponding to the given ID.

Return Values

DVLOC_ERR_NOERROR	The information in the structure is valid.
DVLOC_ERR_WRONG_OPERANDS	Encoded operands do not match the ones expected in the instruction with the given instruction ID.
DVLOC_ERR_PREDICATE_MUST_BE_ZERO	The instructions with the given instruction ID are not predicatable.
DVLOC_ERR_SRC5_MISSING	src5_reg should contain a general register operand.
DVLOC_ERR_INVALID_FLAGS	The flag field contains illegal values.
DVLOC_ERR_INVALID_INSTRUCTION_ID	The instruction ID is not within the bounds of the legal instruction IDs.



CAUTION. *This function, even when it returns* DVLOC_ERR_NOERROR*, does not provide complete assurance that the* AbstrInstrInfo *structure is valid. Be careful when creating* AbstrInstrInfo *structures, as wrong encoding can lead to inaccurate dependency violation detection.*

DVLoc_search_dep_viols_abstr_code()

Searches for dependency violations in an inspection block

```
DVLoc_err_t DVLoc_search_dep_viols_abstr_code(
    DVLoc_inspection_block_t  a_block_type,
    AbstrInstrInfo*  instr_arr,
    long  first_instr_index,
    long  last_instr_index,
    unsigned int*  p_nb_dep_viols);
```

Parameters

a_block_type	Must be DVLOC_INSPECT_INSTR_GROUP.
instr_arr	A pointer to the inspection block.
first_instr_index	The index of the first instruction to be analyzed in the array. See "AbstrInstrInfo - Abstract Instruction" in Chapter 4.
last_instr_index	The index of the last instruction to be analyzed in the array. See "AbstrInstrInfo - Abstract Instruction" in Chapter 4.
p_nb_dep_viols	A pointer to the unsigned integer, where the number of detected dependency violations is placed.

Description

The DVLoc_search_dep_viols_abstr_code() function is relevant only for abstract mode. Use it to send blocks of Intel® Itanium™ architecture code to DVLoc. This function searches for dependency violations in an inspection block defined by *instr_arr*, *first_instr_index*, and *last_instr_index*. It fills in fields in an internal dependency violation database. The dependency violations information is kept in the database until the next function call to DVLoc_execution_clue() or DVLoc_search_dep_viols_abstr_code().

Return Values

DVLOC_ERR_NOERROR	Function completed successfully.
DVLOC_ERR_INVALID_PARAM	Invalid parameter passed to function.
DVLOC_ERR_MEM_EXHAUST	No memory is available. This is a fatal error.
DVLOC_ERR_CLOSED	Library is closed; no services are available.

DVLoc_search_dep_viols()

Searches for dependency violations in an inspection block

```
DVLoc_err_t DVLoc_search_dep_viols(
    DVLoc_inspection_block_t  block_type,
    const unsigned char*  bin,
    Uint64  first_instr_loc,
    Uint64  last_instr_loc,
    unsigned int*  nb_dep_viols);
```

Parameters

block_type	Must be DVLOC_INSPECT_FUNCTION.
bin	A pointer to the inspection block binary.
first_instr_loc	Address of the first instruction to inspect (Address of the first instruction in the inspection block is always 0x0).
last_instr_loc	Address of the last instruction to inspect.
p_nb_dep_viols	A pointer to the unsigned integer, where the number of detected dependency violations is placed.

Description

The DVLoc_search_dep_viols() function is only relevant for static mode. It searches for dependency violations in an inspection block defined by *bin*, *first_instr_loc*, and *last_instr_loc*. It fills in fields in an internal dependency violation database. The dependency violations information is kept in the database until the next function call to either DVLoc_execution_clue() or DVLoc_search_dep_viols().

Return Values

DVLOC_ERR_NOERROR	Function completed successfully.
DVLOC_ERR_INVALID_PARAM	Invalid parameter passed to function.
DVLOC_ERR_MEM_EXHAUST	No memory is available. This is a fatal error.
DVLOC_ERR_CLOSED	Library is closed; no services are available.

DVLoc_search_dep_viols_absolute_loc()

Searches for dependency violations in an inspection block

```
DVLoc_err_t DVLoc_search_dep_viols_absolute_loc(
    DVLoc_inspection_block_t  block_type,
    const unsigned char*  bin,
    Uint64  bin_absolute_loc,
    Uint64  first_instr_loc,
    Uint64  last_instr_loc,
    unsigned int*  nb_dep_viols);
```

Parameters

block_type	Must be DVLOC_INSPECT_INSTR_STREAM.
bin	A pointer to the inspection block binary.
bin_absolute_loc	The virtual address of the binary code being sent.

first_instr_loc	The address of the first instruction to inspect, given that the first instruction in the inspection block is 0x0.
last_instr_loc	The address of the last instruction to inspect, given that the first instruction in the inspection block is 0x0.
p_nb_dep_viols	A pointer to the unsigned integer, where the number of detected dependency violations is placed.

Description

The DVLoc_search_dep_viols_absolute_loc() function is only used in dynamic mode. It searches for dependency violations in an inspection block defined by *bin*, *first_instr_loc*, and *last_instr_loc*. It fills in fields in an internal dependency violation database. The dependency violations information is kept in the database until the next function call to DVLoc_execution_clue() or DVLoc_search_dep_viols_absolute_loc().

Return Values

DVLOC_ERR_NOERROR	Function completed successfully.
DVLOC_ERR_INVALID_PARAM	Invalid parameter passed to function.
DVLOC_ERR_MEM_EXHAUST	No memory is available. This is a fatal error.
DVLOC_ERR_CLOSED	Library is closed; no services are available.

DVLoc_get_dep_viol()

Gets a dependency violation from DVLoc

```
DVLoc_err_t DVLoc_get_dep_viol(
    unsigned int  dep_viol_nb,
    DVLoc_dep_viol_t*  dep_viol)
```

Parameters

dep_viol_nb	Sequential dependency violation number, starting from 0 (input).
dep_viol	A pointer to a dependency violation structure (output).

Description

The DVLoc_get_dep_viol() function fills the structure pointer by *dep_viol* with the information of the dependency violations number indicated by *dep_viol_nb* in the dependency violations database. Accessing a dependency violation beyond the total number of dependency violations currently in the database returns DVLOC_ERR_OUT_OF_RANGE, and the contents of *dep_viol* are cleared.

Return Values

DVLOC_ERR_NOERROR	Function completed successfully.
DVLOC_ERR_OUT_OF_RANGE	Dependency violation number accessed exceeds the total number of dependency violations.
DVLOC_ERR_INVALID_PARAM	Invalid parameter passed to function.
DVLOC_ERR_CLOSED	Library is closed; no services are available.

DVLoc_get_dep_viol_text()

Gets a textual description of a dependency violation

```
DVLoc_err_t DVLoc_get_dep_viol_text(
    unsigned int  dep_viol_nb,
    char*  string,
    unsigned size  string_size)
```

Parameters

dep_viol_nb	Sequential dependency violation number (input)

string	Pointer to a character string (output)
string_size	Size of the character string, in bytes (input).

Description

The DVLoc_get_dep_viol_text() function fills the character string with text information of the n^{th} dependency violation in the database. *n* is determined by *dep_viol_nb*, starting at 0. Accessing a dependency violation beyond the total number of dependency violations currently in the database returns DVLOC_ERR_OUT_OF_RANGE, and *string* is set to a null string.

Return Values

DVLOC_ERR_NOERROR	Function completed successfully.
DVLOC_ERR_OUT_OF_RANGE	Dependency violation number accessed is beyond the total number of dependency violations.
DVLOC_ERR_INVALID_PARAM	Invalid parameter passed to function.
DVLOC_ERR_CLOSED	Library is closed; no services are available.

DVLoc_close()

Closes DVLoc

```
DVLoc_err_t DVLoc_close()
```

Description

The function DVLoc_close() closes DVLoc, freeing the memory that DVLoc internally allocated. The client application may re-open DVLoc after it was closed. Closing DVLoc when it is already closed returns DVLOC_ERR_CLOSED.

Return Values

DVLOC_ERR_NOERROR	Function completed successfully.
DVLOC_ERR_CLOSED	Library is closed; no services are available.

Glossary

abstract instruction	A data structure describing an Intel® Itanium™ architecture instruction. See "AbstrInstrInfo - Abstract Instruction" in Chapter 4.
certain dependency violations	See dependency violations.
Complete Dependency Semantics mode	Full dependency violation detection of all possible semantics.
CFM	Current Frame Marker register.
dependency violation	A violation of data dependency on an Itanium architecture resource, resulting from two instructions in the same instruction group accessing the same resource. All types of dependency violation are detected: • WAW dependency violation Write After Write • RAW dependency violation Read After Write • WAR dependency violation Write After Read (rare) As a static analyzer, DVLoc lacks the run-time resource values. DVLoc provides an API through which the client can pass information about run-time values, allowing more accurate dependency violation detection. See Chapter 5, "Clues". DVLoc reports dependency violations as one of the following: • certain dependency violations, which do not depend on resource values at run time

	• potential dependency violations, whose actual occurrence depends on resource values at run time. For example, in the following code the WAW dependency violation on GR4 exists only if P1 and P2 are both true at run time: `(p1) mov r4=r5` `(p2) mov r4=r6`
entry point	A point at which the control flow might change; for example, a function label or a target of a direct branch.
inspection block	Name of the block of Itanium architecture code sent to DVLoc for dependency violation detection. In abstract mode, the block is an array of abstract instructions. In static and dynamic modes, the block is a block of Itanium architecture binary code.
instruction ID	Enumeration of type `Inst_id_t`. See "Instr_id_t" in Chapter 3.
mutex	A group of predicate registers are said to be mutex in a region of code, if at any execution of the region, only one register of the group can be set.
Piig violation	Some instructions are required to be in a specific place in an instruction group. A Place-in-instruction-group (Piig) violation, is when an instruction is not in the correct position in an instruction group. Types of Piig violations: • Instructions that must be first in an instruction group and are not; for example, `alloc`. • Instructions that must be last in an instruction group and are not; for example, `cover`.
potential dependency violations	See dependency violations.

predicate registers relation	A logical connection between the values of several predicate registers. The supported relations in DVLoc are mutex and implication.
SAC	Safe across calls.
scope	The lifetime of the information given by the clue.
resource	A register or part of a register. Examples: • General register is a resource. • `PSR.be` is a resource. • `PSR.sm` is a resource. All the Itanium architecture resources are enumerated in the `sched_IA64.h` header file.

Index

Symbols

__int64, 2-2, 4-1

A

abstract instructions, 2-3, Glossary-1
abstract mode, 1-1, 2-2, 3-1, 4-3, 6-1
allocated memory, 2-4, 2-7, 2-8

B

binary code, 2-6, 2-8
block types, 3-2
bundling, 1-1

C

capabilities, 2-1
CFM, 2-3, 2-6, Glossary-1
clear, 5-3
clear predicate register relations, 5-6
close DVLoc, 2-4, 2-7, 2-8, 6-10
clue timing, 3-11, 5-2
clues, 2-3, 2-6, 2-7, 5-1
 types, 3-9
code life cycle, 1-1
code partitioning, 2-1
command-line option, 2-2
Complete Dependency Semantics mode, 6-2, Glossary-1
conditional branches, 2-4
control flow, Glossary-2
Current Frame Marker, 2-3, Glossary-1

D

data, 5-17
data semantic, 3-8
data structures, 4-1
debug mode, 6-4
Dependency Violation Locator, 1-1
dependency violations, 1-1, 2-1, 3-7, Glossary-1
 IDs, 3-4
 semantics, 3-8
 types, 3-4
documentation, 1-2
DVLoc, 1-1
DVLoc modes, 2-1
-DWINNT, 2-2
dynamic mode, 1-1, 2-2, 2-7, 3-1, 6-1

E

entry point, 2-4, Glossary-2
environment, 2-1
 settings, 3-2
 variables, 6-2
execution, 1-1
 clue, 6-3

F

final, 1-1
fonts, 1-2
functions, 3-5, 6-1

G

Glossary, 1-2

I

implication, 5-3, 5-7
implied semantic, 3-8
ImpliedF semantic, 3-8
indirect register files, 2-8, 5-16
initialize DVLoc, 6-1
inspection block, 2-3, 2-6, 2-7, 5-1, 6-5, Glossary-2
instr semantic, 3-8
instructions, 3-3
 ID, Glossary-2

L

ld8.fill instruction, 2-3, 2-6, 2-8
life cycle, 1-1
location, 4-2, 4-3, 5-2

M

macro, 5-10
modes, 2-1, 3-1, 6-1
modulo-scheduled branches, 2-4
mutex, 5-3, 5-6, Glossary-2

O

open DVLoc, 2-2, 2-5, 2-7, 6-1

P

Piig violations, 2-1, Glossary-2
pmd, 2-8, 5-16
predicate registers, 3-12, 3-13, 5-9
 relation, 5-6, Glossary-3
predicate relations, 5-3
publications, 1-2

R

register operands, 3-4
register values clue, 5-15
resources, 3-3, 4-1, Glossary-3
 dependencies, 2-1
return values, 3-5
rotating region size, 5-13
rotating register, 5-13

S

SAC, 3-10, 5-12, Glossary-3
safe across calls, 3-10, 5-12, Glossary-3
sched_enums.h, 2-1
sched_IA64.h, 2-1
Scheduling Library, 1-1, 2-1
scopes, 2-6, Glossary-3
search dependency violations, 6-6
search function, 2-3, 2-6, 2-8
search inspection block, 6-5
semantics, 3-8, Glossary-1
send block, 2-3, 2-6, 2-8
send clues, 2-3, 2-6, 2-7
set, 5-3
set of relations approach, 5-8
single relation approach, 5-4
singleton, 5-3
skip dependency violations clue, 5-17
src5_reg, 3-7, 4-4

st8.fill clue, 5-14
st8.spill clue, 5-14
st8.spill instruction, 2-3, 2-6, 2-8
static mode, 1-1, 2-2, 2-5, 3-1, 6-1
stops, 2-4

T

timing, 3-11, 5-2
types of clues, 3-9
types of dependency violations, 3-4

U

Uint64, 2-2, 4-1
UNAT, 2-3, 5-14
unconditional branches, 2-4
unconditional compare, 5-7
usage flows, 2-1

V

validate, 6-4